John von Neumann Institute for Computing

Runtime Checking of MPI Applications with MARMOT

B. Krammer, M.S. Müller, M.M. Resch

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 893-900, 2006.

http://www.fz-juelich.de/nic-series/volume33

Runtime Checking of MPI Applications with MARMOT

Bettina Krammer[a], Matthias S. Müller[b] and Michael M. Resch[a]

[a]High Performance Computing Center Stuttgart (HLRS)
Allmandring 30, D-70550 Stuttgart, Germany
{krammer, resch}@hlrs.de

[b]Centre for Information Services and High Performance Computing (ZIH)
D-01062 Dresden, Germany
matthias.mueller@tu-dresden.de

The Message Passing Interface (MPI) is widely used to write parallel programs using message passing, but it does not guarantee portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler/architecture/MPI implementation. In many cases the problem is a subtle programming error in the application undetected on the platforms used previously. Finding this bug can be a very strenuous and difficult task. This paper presents MARMOT, an automated tool designed to check the correctness of MPI applications during runtime. Examples of such violations are the introduction of irreproducibility, deadlocks, incorrect management of resources such as communicators, groups, datatypes etc. or the use of non-portable constructs.

1. How to Find Bugs in Parallel Programms

Parallel programs can not only be inflicted with all the bugs known from serial programming (multiplied by the number of processes) but also by bugs resulting from the interaction of several parallel processes. Reproducibility is often not given. Finding these bugs in a complex parallel application is quite a painful task. Fortunately there are powerful tools for the different aspects of debugging, e.g. tools for memory checking or for correctness checking. Apart from the classical way of debugging – `printf` statements – the different solutions are roughly grouped into four different approaches: classical debuggers, special MPI libraries and other tools that may perform a runtime or post-mortem analysis.

1. The freely available debugger gdb [16], which is also used with its graphical front-end ddd [17], has currently no support for MPI, but it can be attached to one or several, possibly already running MPI processes. The same can be done with special memory-checking debuggers like valgrind [18,19]. More convenient are parallel debuggers, which are based on serial debuggers like gdb. They provide the usual interactive functionality of debuggers, such as single-stepping, breakpointing, evaluating variables, etc., but additionally allow the user to monitor and act on groups of processes in a single debugging session. Examples are the well-known commercial debuggers Totalview [14] or DDT [13]. These debuggers can also be used for a post-mortem analysis of core files.

2. The second approach is to provide a special debug version of the MPI library (e.g. mpich or NEC-MPI). This version is not only used to catch internal errors in the MPI library, but also to detect some incorrect usage of MPI by the user, e.g. a type mismatch of sending and receiving messages or mismatched collective operations [4–6].

3. Another possibility is to develop tools dedicated to finding problems within MPI applications at runtime. At present, three different message-checking tools are under more or less active development: MPI-CHECK [8], Umpire [3] and MARMOT [9,10]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems like deadlocks [8]. Like MARMOT, Umpire [3] uses the profiling interface.

4. The fourth approach is to perform a post-mortem analysis by collecting all information on MPI calls in a trace file. After program execution, this trace file is analysed by a separate tool or compared with the results from previous runs [7]. An example of this is the Intel Message Checker (IMC) [20]. This approach is also used by many tools with respect to performance analysis, and indeed, in some cases it can be very enlightening to "abuse" a performance tool for debugging.

As no tool is an all-in-one device suitable for every purpose, a combination of different tools will probably aid the developers most. While a memory-checking debugger may be able to diagnose that an application crashes due to an uninitialized variable, it will definitely not help you much in finding incorrect usage of the MPI interface as MARMOT does. Regardless, not every error can be caught by tools.

2. Architecture of MARMOT

MARMOT uses the so-called profiling interface to intercept MPI calls and analyse them during runtime. It does not require any modification of the application's source code nor of the MPI library, it is just a library that has to be linked to the application in addition to the underlying MPI implementation.

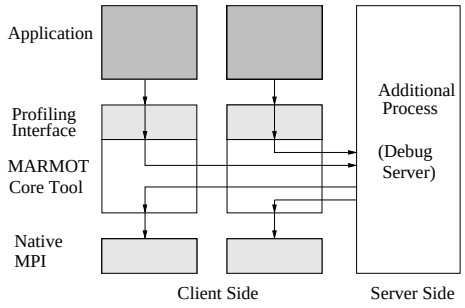

Figure 1. Design of MARMOT.

Figure 1 illustrates the design of MARMOT. For all tasks that require a global view, e.g. deadlock detection or the control of the execution flow, MARMOT uses an additional process, the so-called Debug Server. Each client registers at the debug server, which in turn gives its clients the permission for execution in a roundrobin way. In order to ensure that this additional debug process is transparent to the application, we map `MPI_COMM_WORLD` to a MARMOT communicator containing only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they automatically exclude the debug server process. Everything that can be checked locally, e.g. verification of arguments such as tags, communicators or ranks, is performed by the client. Additionally,

the clients and the Debug Server use MPI internally to transfer information. This server/client architecture inflicts a bottleneck, thus affecting the scalability and performance of the tool, especially for communication-intensive applications [12].

3. Runtime Checking of MPI Applications

3.1. Runtime versus Post-mortem Analysis

Compared to human intervention, automatic checking has the potential to scale better. Two different scalability issues are important. First, scalability to many processors, and second, scalability in runtime. We see programs that show illegal or non-portable use of MPI only after an exceptionally long runtime. For instance, an application increases the send/receive tag after each iteration step by 1. After thousands of iterations, the tag finally exceeds the limit of up to 32767 that is guaranteed by the MPI standard. While many MPI implementations grant a much higher range of tags, there are implementations that are strict about that limit, for example versions of LAM-MPI prior to 7.0.

Since runtime checking does not require storing a huge amount of persistent information there is no scalability limit regarding runtime. Another advantage of runtime checking, compared to offline analysis, is that the application is still up and running when an error is detected. Thus other tools can be used to further analyse the situation. For example, a debugger can be attached to check the content of variables and understand the status of the application.

3.2. Frequent Problems of Parallel Programming

Parallel programming is a complex challenge. It offers enough pitfalls that MPI can imaginably stand for “Maddening Programming Interface”. Among the Top Ten common programming errors are:

- **Deadlocks:** MARMOT contains a mechanism to automatically detect deadlocks and notify the user where and why they have occurred. In general, deadlocks are caused by the non-occurrence of something else, for example mismatched send/receive operations or mismatched collective calls. One can distinguish between *real* deadlocks, which occur inevitably, and *potential* deadlocks, which may occur only under certain circumstances, e.g. depending on data races or on the implementation, for instance, if a standard send is implemented as a buffered send or not. In this code snippet process 0 and process 1 exchange messages between each other.

```
if (rank == 0) {
  // send to 1 and receive from 1
  MPI_Send(...);
  MPI_Recv(...);
} else if (rank == 1) {
  // send to 0 and receive from 0
  MPI_Send(...);
  MPI_Recv(...);
}
```

  If the `MPI_Send` is implemented in buffered mode, for example for small message sizes, this code will not deadlock, otherwise it will. Currently MARMOT’s deadlock detection is based on a timeout mechanism and therefore finds all real deadlocks. MARMOT’s debug server

surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning. The user is then able to trace the last few calls on each node. It is also possible that attaching MARMOT (or any other tool) to an application slightly changes the execution flow in such a way that a potential deadlock becomes apparent.

- **Data races:** Potential race conditions can be caused by various reasons, e.g. by the use of a receive call with the wildcard `MPI_ANY_SOURCE` as source argument or the wildcard `MPI_ANY_TAG` as tag argument, by the use of random numbers, or by the fact that nodes do not behave exactly the same. Some users also rely on collective calls being synchronising, however, the only synchronising collective call is the `MPI_Barrier`. Other collective calls can be synchronising or not, depending on their implementation. For example, assume that any of the send calls on the processes 1 and 2 match to any of the receive calls on process 0.

```
if (rank == 0) {
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
  MPI_Bcast(....);
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
} else if (rank == 1) {
  MPI_Send(...);
  MPI_Bcast(...);
} else if (rank == 2){
  MPI_Bcast(...);
  MPI_Send(...);
}
```

  If the `MPI_Bcast` is synchronising process 0 will have to receive the message from process 1 first. If it is not then the message order will not be deterministic: either the message from process 1 or from process 2 can be received first. At present, MARMOT indicates the use of wildcards, but it does not construct dependency graphs to view the different possible executions nor does it use methods like record and replay to identify and track down bugs in parallel programs [7] or to compare different runs. Why does one need a tool to detect this sort of argument as a simple *grep* command on the source code would give the same result? Actually, a search command does neither show the execution flow nor will it be able to detect this argument if the application takes functions from some other library with hidden MPI calls.

- **Mismatches:** Mismatches in arguments of one call can be detected locally and are sometimes even detected by the compiler. Examples are wrong type or number of arguments. Mismatches are also seen in arguments involving more than one call, e.g. in send/receive pairs or in collective calls. Special attention is needed when comparing matched pairs of derived datatypes because it is legal to send, for example two (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`), or to send one (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`) (a so-called *partial* receive). MPI implementations usually abort an application when there is a datatype mismatch, e.g. send an `MPI_INT` and receive an `MPI_DOUBLE`, but no exact diagnosis of the mismatch is given.

- **Resource handling:** This is an area in MPI where incorrect usage may result in fatal errors with almost no obvious link to the real cause. Since they are very difficult to find, we place special focus into detecting them. MARMOT is able to keep track of the proper construction, usage and destruction of all MPI resources, such as communicators, groups, datatypes, etc. As these resources are "opaque" objects and therefore implementation-dependent, MARMOT has its own book-keeping of these resources and, thus, duplicates the management done by the underlying MPI library. MARMOT also checks if requests and other arguments (tags, ranks, etc.) are used correctly, e.g. if an active requests is reused. The main functionality is implemented for the C language binding, whereas the functionality for the Fortran language binding is obtained through a wrapper to the C interface. Special attention is paid to the verification of the datatypes because they are one of the major differences between the C and the Fortran language binding.

- **Memory and other resource exhaustion:** Non-blocking calls such as `MPI_Isend` etc. can complete without issuing a matching test or wait call. However, the number of available request handles is limited (and implementation defined). Therefore requests should always be freed, as should allocated communicators, datatypes, etc. MARMOT gives a warning when a request is reused, and also when there are active or non-freed requests left at the `MPI_Finalize`. Another issue is reusing memory that is still in use, for example by reading/writing from/into a buffer by an unfinished send/receive operation. MARMOT does currently not perform any checks if a buffer can be reused safely because the transmission of data has completed. This kind of check is a subtle task that requires some insight into an MPI implementation: what is really going on when calling e.g. `MPI_Issend` or `MPI_Irecv`, how does that depend on the message size, etc.? In some cases, MARMOT checks if buffers are overwritten by mistake, e.g. for `MPI_Gatherv` and similar collective calls, it is verified if on the root process data is overridden due to an erroneous array of displacements.

- **Portability:** The MPI standard leaves many decisions to the implementors, for example how to implement opaque objects and handles to these objects, if to implement `MPI_Send` as buffered call or not, if to implement collective calls as synchronising calls, if to make the implementation thread-safe or not, etc. Some of these issues can already be detected at compile time when the application is ported to another environment, some can be found at runtime by MARMOT, e.g. using a tag beyond the guaranteed limit. Another example of non-portable constructs can even be found in the MPI-1 standard on pages 79 - 80 (and we have found it exactly like that in a user's code):

```
 /* build datatype describing structure */
 ...
 MPI_Aint disp[3];
 int      base;

/* compute displacements of structure components */
...
base = disp[0];
for (i=0; i<3; i++) disp[i] -= base;

MPI_Type_struct(...,disp,...);
```

If you ever try to use the datatype constructed above on a 64-bit-architecture the code will probably crash without any warning, not at the construction step, but later on when the datatype is used, e.g. in a send/receive call. The reason is that `MPI_Aint` is `long` and not `int` there, and thus, some significant bits are lost in the computed array of displacements.

MARMOT supports the complete MPI-1.2 standard, although not all possible tests (such as consistency checks) are implemented. It can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behaviour is not indicated by the implementation itself, nor are checks performed that would decrease the performance too much, such as consistency checks. What is worse, MPI implementations tolerate quite a few errors without warnings or crashing, by simply giving wrong results.

MARMOT is tested on Linux Clusters with IA32/IA64 processors, Cray, Hitachi, IBM Regatta and NEC SX systems, using different compilers (GNU, Intel, PGI,etc.) and different MPI implementations (mpich, LAM/MPI, vendor MPIs, etc.). Functionality and performance tests are performed with test suites, microbenchmarks and real applications [11,12].

4. More Examples

There are many examples of errors that are tolerated by MPI implementations or that only occur on specific platforms, or occur under specific circumstances. For example, in mpich it is possible to use the Fortran datatype `MPI_INTEGER` in a C program without any problems or warnings because, in this implementation, some Fortran datatypes are defined in the C include file `mpi.h`. Implementations that are stricter, e.g. LAM/MPI, will not execute that code. When analysing the development version of a medical application that uses a 3D Lattice-Boltzmann method for blood flow calculation, we find another portability problem of that kind. In many places the developers equate `MPI_Comm` with `int`. This is a dangerous thing to do because, in mpich, the opaque object `MPI_Comm` is actually defined as an `int` and therefore the code works without a problem. However, in LAM/MPI, `MPI_Comm` is defined as a pointer to a `struct` and therefore it breaks on any platform where a pointer does not fit into an integer.

When we test this application with different input files representing the geometry of the artery we find other problems. In the simplest case, a mere tube with an approximately constant radius, the code runs without any problems. When calculating the blood flow for an artery stenosis, i.e. using a tube with varying radius, the application stalls and MARMOT finds a deadlock caused by process 0, which performs an `MPI_Sendrecv` whereas all other processes perform an `MPI_Bcast`. A very simplified skeleton of the source code shows why:

```
main {
  ...
  // compute number of iterations depending on the radius
  if (radius < x) num_iter = y; else num_iter = z;

  for (i=0; i < num_iter; i++)
  {
    // compute blood flow and exchange results
    // with neighbours using MPI_Sendrecv
```

```
      computeBloodflow(...);
    }

    // communicate results using MPI_Bcast
    writeResults(...);
  }
```

Every process calculates its own number of iterations depending on the radius, but unfortunately, they do not communicate to agree on a maximum number of iterations. As a result, process 0 tries to perform more iterations having a piece of the artery with a bigger radius than the others, and therefore tries to exchange results with its neighbour using the `MPI_Sendrecv` whereas the others already have finished their iterations and try to communicate with the `MPI_Bcast`. This shows how important it is to choose relevant input data sets for an effective runtime checking. It also shows how difficult it is for developers to keep track of all the MPI communication when it is hidden in subroutines.

Another input file representing a forked artery reveals yet another programming error. In this case, every process results in different values for the send/receive counts in the collective call `MPI_Gather`. On some platforms the application runs without a problem, but on some platforms the different values cause a segmentation violation. Strangely enough, on one platform the application runs without a problem, but when attaching a performance analysis tool to it, it crashes. It appears to be the fault of the tool, but in reality there is a bug in the application. Antithetically, it is possible that bugs never occur in the presence of tools (so-called *Heisenbugs*).

5. Conclusions and Future Work

Hitherto, we have presented the MARMOT tool, which analyses the behaviour of an MPI application during runtime and checks for errors frequently made in the use of the MPI API. The functionality of this tool has been tested with real world applications.

Although there is still plenty of work to do, we believe that MARMOT is on the right track and will become an indispensable tool in the development of MPI-parallel applications. Future work includes an extension of MARMOT's functionality according to the users' needs. To offer a more user-friendly interface, to support frequently used parts of MPI-2 such as parallel file I/O [10] or to support hybrid applications written in OpenMP and MPI. Another goal is to improve the performance and scalability of the tool, especially for communication-intensive applications.

Acknowledgments

The development of MARMOT was partially supported by the European Union through the IST-2001-32243 project "CrossGrid".

References

[1] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org/`.

[2] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. `http://www.mpi-forum.org/`.

[3] J.S. Vetter and B.R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, Texas, 2000.

[4] William D. Gropp. Runtime Checking Of Datatype Signatures In MPI. In *Recent Advances In Parallel Virtual Machine And Message Passing. 7th European PVM/MPI Users' Group Meeting. LNCS 1908*, pages 160-167. Springer 2000.

[5] Chris Falzone, Anthony Chan, Ewing Lusk and William Gropp. Collective Error Detection for MPI Collective Operations. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 138-147. Springer 2005.

[6] J.L. Träff and J. Worringen. Verifying Collective MPI Calls. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 18 - 27, Springer, 2004.

[7] Dieter Kranzlmüller. *Event Graph Analysis For Debugging Massively Parallel Programs.* Phd thesis, Joh. Kepler University Linz, Austria, 2000.

[8] Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra and Marina Kraeva. Deadlock Detection In MPI Programs. In *Concurrency and Computation: Practice and Experience.* 2002, vol. 14, pages 911 - 932.

[9] MARMOT. `http://www.hlrs.de/organization/tsc/projects/marmot`

[10] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI I/O Analysis and Error Detection with MARMOT. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 242 - 250, Springer, 2004.

[11] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Proceedings of PARCO 2003*, pages 493-500, Elsevier, 2004.

[12] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI Application Development Using the Analysis Tool MARMOT, In *Proceedings of ICCS 2004, LNCS 3038*, pages 464 - 471, Springer 2004.

[13] DDT. The Distributed Debugging Tool.
`http://www.streamline-computing.com/softwaredivision_1.shtml`

[14] Totalview. `http://www.etnus.com/Products/TotalView`

[15] mpigdb.
`http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm#Node29`

[16] The GNU Project Debugger. `http://www.gnu.org/manual/gdb`

[17] The Data Display Debugger. `http://www.gnu.org/software/ddd/`

[18] Brett Carson and Ian A. Mason. ClusterGrind: Valgrinding LAM/MPI Applications. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 325-332. Springer 2005.

[19] Valgrind support for MPIch.
`http://www.hlrs.de/people/keller/MPI/mpich_valgrind.html`

[20] Jayant DeSouza, Bob Kuhn and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. *SE-HPCS '05*, St. Louis, Missouri, USA. `http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf`

[21] A. Tirado-Ramos, H. Ragas, D. Shamonin, H. Rosmanith, and D. Kranzlmueller. Integration of blood flow visualization on the grid: the flowfish/gvk approach. In *2nd European Across Grids Conference*, Nicosia, Cyprus, January 28-30 2004.